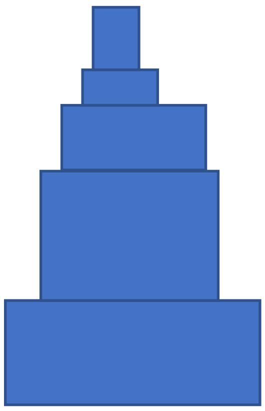
Box-Stacking Problem

The Box Stacking Problem is an optimization problem involving a set of three-dimensional rectangular boxes (cuboids), each defined by length (l), width (w), and height (h). The objective is to maximize the height of the stack constructed by arranging these boxes.
The boxes may be rotated in any way possible to be used as a base, but the dimensions of the base of the box of a lower level must be greater than the dimensions of the base of the box which would be placed on top of the lower box (so that the smaller-base box can be comfortably placed on a larger-base box to build up the rising pile of boxes). This would ensure stability for the pile of boxes.



This problem is closely associated with the Longest Increasing Subsequence problem due to its focus on finding an optimal ordering under specific constraints.
For reference, the basic Longest Increasing Subsequence (LIS) problem involves finding the longest subsequence in a given sequence of numbers where the subsequence is strictly increasing. A subsequence is defined as a sequence derived from the original sequence by deleting some or no elements, without changing the order of the remaining elements.
For ex:- for the group of numbers: 5, 28, 4, 42, 20, 34, 49, 47, 55.
The LIS will be: 5, 28, 42, 49, 55. The length of the LIS will be 5.
So here, the 'specific constraint' is that the order shouldn't be changed, and every element must be greater than its predecessor. For the Box Stacking problem, the constraint lies in the base sizes of the boxes.

Analysing and solving the Box Stacking problem has significant applications in logistics and construction, industrial design, robotics and various other related fields where efficient space utilization and stability are essential.
Here, we explore several algorithmic approaches for solving this problem, comparing their efficiency and effectiveness in achieving optimal stacking heights. By examining these algorithms, we aim to highlight practical methods for tackling similar multi-dimensional optimization problems in real-world applications.

Problem and Constraints:

Each box can be represented in three forms, based on three orientations: (l, w, h) or (w, h, l) or (h, l, w), depending on what two dimensions (considering first and second in the ordered triples for simplicity) become the base.
Note that the dimensions l, w, h are just labels here though. So, say for a box, the length l is 5 cm. This 5 cm edge can serve as the height h too if it's required in the situation (based on the base of the lower box).
If the box with base area B_i is placed on box with base area B_j, then $B_i < B_j$.
And also, we have to try to maximise the total height H, where $H = \sum h$. [i.e. the summation of the h's (these h's will basically be the heights obtained from the stacked boxes)].

Solutions:

There can be different approaches to finding solutions to this problem.

<u>Recursive Solution:</u>
Here, we generate all possible rotations and look at each configuration for all the boxes.
We simplify by sorting the base areas (l*w) of the boxes in descending order (checking from larger-base boxes to smaller-base boxes).
Now, say H(i) is the maximum achievable height with the i-th box as the top-most/current box.
So then,

$$\mathbf{H(i) = h_i + \max_{j<i}\{H(j) \mid l_j > l_i \text{ and } w_j > w_i\}.}$$

Here;
h_i : the height of the box i,
H(j) : the maximum height of a stack with j-th box as the topmost box,
$l_j > l_i$ and $w_j > w_i$: depicts the condition for stable stacking.

So here for box i, the maximum stack height (H(i)) is its own height (h_i) plus the maximum height achievable by stacking 'valid' ($l_j > l_i$ and $w_j > w_i$) boxes below it.
Ex:
Say we have three boxes with dimensions (1,2,3) , (3,4,5) , (6,7,8).
So, we have 9 total possible rotations for these boxes:
(1,2,3),(2,3,1),(3,1,2) ; (3,4,5),(4,5,3),(5,3,4) ; (6,7,8),(7,8,6),(8,6,7).
Descending areas for the bases for these configurations:
(7,8,6): Area = 7×8=56
(8,6,7): Area = 8×6=48
(6,7,8): Area = 6×7=42
(4,5,3): Area = 4×5=20
(5,3,4): Area = 5×3=15
(3,4,5): Area = 3×4=12
(2,3,1): Area = 2×3=6
(3,1,2): Area = 3×1=3
(1,2,3): Area = 1×2=2

So,
As Box (7,8,6) has the largest area, it becomes the base. Here, H(1) (for box 1) is 6.
Box (8,6,7) becomes the base too, no box would go below it. Here, H(2) = 7.

Similarly again, no previous considered box has a larger base (as we already arranged the bases in a descending order). So, (6,7,8) would also be considered as the base. Here, H(3) = 8.
Now, while considering box (4,5,3), we notice that one of the first three configurations we observed would be below it (as it has larger base).
So, overall max height of the stack = H(4) = h_4 + max{H(1), H(2), H(3)} = 3 + 8 = 11.
Similarly for (5,3,4), max height for the stack = 4 + 8 = 12.
Similarly for (3,4,5), max height for the stack = 5 + 8 = 13.
Now, we arrive at our final box and consider its final three configurations.
For (2,3,1), max height for the stack = 1 + max{11,12,13} = 1 + 13 = 14
For (3,1,2), max height for the stack = 2 + 13 = 15
For (1,2,3), max height for the stack = 3 + 13 = 16.

So, the maximum stack height achievable with the given boxes is 16 units.
Note: here, the initial values (like H(1)) will be calculated multiple times. They aren't shown here repeatedly to avoid redundancy.

The recursive solution examines all subsets of boxes and considers each valid stacking arrangement. While considering any and every box, either the box is included in the stack, or excluded as it doesn't meet the requirements. Also, each box can have up to 3 rotations to check. The number of possible configurations therefore rise exponentially. The time complexity for the recursive solution is O(2^n) in the worst case (as for n boxes, 2^n possible subsets can be there – either include or exclude; along with it becoming 3n rotations too, for n boxes). As we saw in the example above, many configurations are unnecessarily computed repeatedly in this approach, which can be optimised and made better by considering dynamic programming.
The space complexity for this approach is O(n), as the recursion tree explores up to n levels.

The recursive approach is simple to understand and implement, and would give optimal results as it examines all the possible configurations. It serves as a great fundamental framework for developing other approaches too. But due to it having redundant computations it isn't very efficient for a large amount of inputs.

Dynamic Programming Solution:
The Dynamic Programming (DP) approach to this problem builds up on the recursive solution approach by optimising it. It stores intermediate results to avoid redundant computations. The DP approach, similar to the recursive approach, builds a solution iteratively by sorting the boxes by base area and computing the maximum height for each box as the top box in the stack.
A DP Array is initialized: dp[i] = h_i, as the stack with only the i-th box, which has height h_i.
For each box i, where for all j < i, $l_j > l_i$ and $w_j > w_i$:
dp[i] = max (dp[i], h_i + dp[j])
The result is the maximum value in the dp array:
Max Height = $\max_i$\{dp[i]\}
Ex:
Say again we have the three boxes we previously considered, with dimensions (1,2,3) , (3,4,5) , (6,7,8).
So again we generate the 9 different rotations possible for the boxes, and sort them by base area in decreasing order.

SNo.	w	l	h / h_i / dp[i] initialization	Base Area
1	7	8	6	58
2	6	8	7	48
3	6	7	8	42
4	4	5	3	20
5	3	5	4	15
6	3	4	5	12
7	2	3	1	6
8	1	3	2	3
9	1	2	3	2

Now, we start stacking boxes:
dp[i] = max (dp[i], h_i + dp[j])

SNo.	h_i	Base Area	Boxes which Support	Updated dp[i]
1	6	58	None (Base)	6
2	7	48	None (Base)	7
3	8	42	None (Base)	8
4	3	20	1,2,3	3 + 8 = 11
5	4	15	1,2,3	4 + 8 = 12
6	5	12	1,2,3	5 + 8 = 13
7	1	6	1,2,3,4,5,6	1 + 13 = 14
8	2	3	1,2,3,4,5,6	2 + 13 = 15
9	3	2	1,2,3,4,5,6	3 + 13 = 16

So, Maximum Height obtained = 16 units.
As we observe, the basic idea behind both the Recursive and the Dynamic Programming approaches is the same. The difference lies in their computational methods and the time taken during that process.

The DP method proves to be a faster and more practical approach in comparison to the recursive one for medium to large number of inputs. Managing through overlapping subproblems and optimal substructures yields efficient optimal solutions.
The time complexity reduces to O(n^2), as now there isn't an exponential check for every box, and just for the total 3n rotations (followed by their sorting and computation). The space complexity remains O(n).

Greedy Solution:
The Greedy Approach for the Box Stacking problem works by attempting to select the box with the largest possible area at each step, and ensuring the boxes are stacked in decreasing order of their base area. This is a trivial and a very fast approach in comparison to the other approaches discussed, which will work well and accurately possibly for only a very small amount of inputs, and that too isn't guaranteed!
As this approach only prioritizes following the constraints and makes local decisions, it does not consider all the possible configurations. It doesn't factor in the overall problem. And so,

this approach will not always yield an optimal solution, and may only be used to find an approximate solution.

Ex:
Again we have the three boxes we previously considered, with dimensions (1,2,3) , (3,4,5) , (6,7,8).
Now, arranging the possible rotation configurations based on descending order of base area:

1. (7,8,6): Area = 7×8=56
2. (8,6,7): Area = 8×6=48
3. (6,7,8): Area = 6×7=42
4. (4,5,3): Area = 4×5=20
5. (5,3,4): Area = 5×3=15
6. (3,4,5): Area = 3×4=12
7. (2,3,1): Area = 2×3=6
8. (3,1,2): Area = 3×1=3
9. (1,2,3): Area = 1×2=2

So, we start with the box with the largest base, 1, i.e. (7,8,6). Height = 6. Current Stack = {1}
We observe that as soon as the largest possible base was observed, it was immediately selected, and put in the stack. And so, this automatically rules out the 2nd and 3rd configuration (Box 1).
We now move on to the 4th configuration (Box 2), (4,5,3). As the base area 20 < 56, it's instantly selected. So, overall height = 6 + 3 = 9. Current Stack = {1, 4}. Again this rules out the other Box 2 configurations, i.e. the 5th and 6th configuration.
Now, coming on to the 7th configuration (Box 3), (2,3,1). As the base area 6 < 20, it's instantly selected.
So, max final height achievable using this approach = 9 + 1 = 10.
Final stack = {1, 4, 7}
And again as the 7th configuration was instantly selected, it immediately rules out the 8th and 9th configuration of Box 3.
So, here we can clearly see, we got the non-optimal maximum height of 10 units.

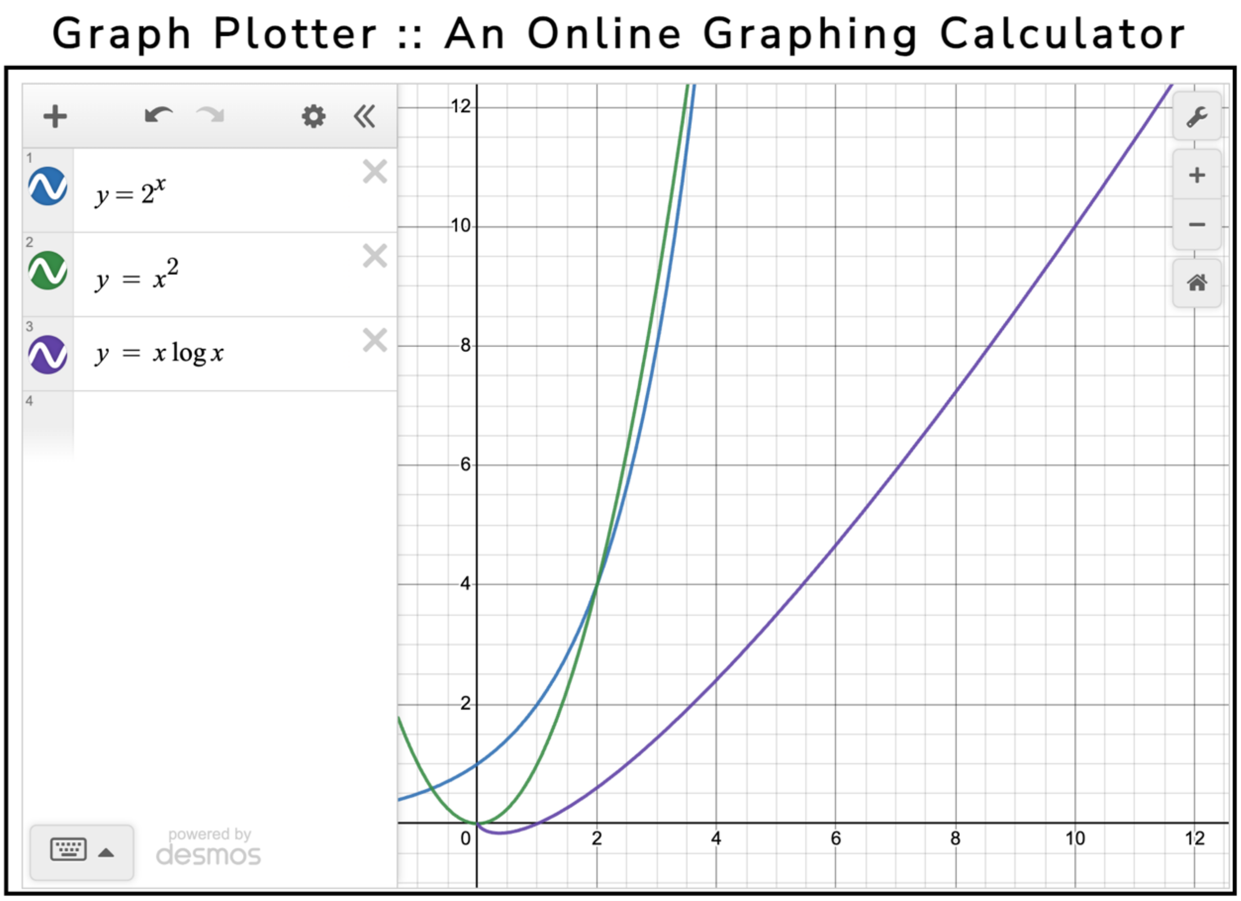
Now, one may say, that if this is a greedy algorithm, it should always prioritise stacking of boxes, in order to reach the maximum height for the stack. But then by prioritizing for greater heights, it would ignore the base area, and would break the base area constraint. Greater heights for the boxes is of course needed and getting that would be the optimal choice for every box, but maintaining the base area constraint is a must for every situation in the Box Stacking problem, and therefore must be prioritized over greater heights.

The time complexity is lesser than the previously discussed approaches, at O(nlogn), as only sorting dominates the overall complexity. The space complexity remains O(n).
Pseudocode for the Greedy Approach:

```
Input: List of boxes with dimensions (l, w, h)
Output: Maximum height of the stack using greedy approach

1. Generate all rotations for each box:
   - For each box with dimensions (l, w, h), create:
     - (l, w, h), (w, h, l), (h, l, w)
2. Sort all rotations in descending order of base area:
  - Base area = length * width
3. Initialize an empty stack.
  - CurrentHeight = 0
4. For each box in the sorted rotations:
  - If the box can be placed on top of the stack:
      - Add the box to the stack
      - Update CurrentHeight += box.height
5. Return CurrentHeight
```

Comparison:



A graph for the time complexities of the three solution approaches.
The Blue exponential curve depicts the Recursive Solution approach.
The Green quadratic curve depicts the Dynamic Programming Solution approach.
And finally, the Purple log-linear depicts the Greedy Solution approach.

Some More Approaches and Real-Life Examples:

For large-scale instances of the Box Stacking problem, where an exact solution may be computationally expensive; approximate or heuristic algorithms are often used. These approaches (like the greedy approach) aim to provide near-optimal solutions in significantly less time, making them ideal for real-time or large-scale applications. Some of these are:

Genetic Algorithm (GA):

Genetic algorithms are inspired by the principles of natural selection and genetics. They are particularly useful for combinatorial optimization problems like Box Stacking. A GA can be applied to the Box Stacking problem in a number of steps.
Each chromosome represents a possible stacking configuration of boxes. A chromosome can be encoded as a permutation of box indices, where the order determines how the boxes are stacked. There's an initial population of random chromosomes (stacking configurations) that is generated. And, there's a fitness function which is basically a function to evaluate the quality of each chromosome. For the Box Stacking problem, the fitness function could be the total height of the stack represented by the chromosome, ensuring that stacking constraints (base area conditions) are followed. Then selection happens, in which pairs of chromosomes are selected from the population based on their fitness. Fitter chromosomes have a higher chance of being selected. Common methods include roulette wheel selection or tournament selection. There's crossover, which combines two parent chromosomes to produce offspring. A crossover operation mixes the stacking order of boxes between two parents. There's mutation, which introduces small random changes to offspring to maintain diversity in the population. And then finally there is termination. Repeat selection, crossover, and mutation occurs until a termination condition is met (e.g., a fixed number of generations or no significant improvement in fitness).
Using this approach yields near-optimal solutions efficiently.

Ant Colony Optimization (ACO):

Ant Colony Optimization is inspired by the foraging behavior of ants. It uses a population of artificial "ants" to explore the solution space and find optimal or near-optimal solutions. Like GA, this occurs in a number of steps as well.
Firstly, the whole problem may be represented as a graph, where each node corresponds to a box (or a rotation of a box). Edges between nodes represent the transition from one box to another in the stacking order. A Pheromone value for each edge must be maintained, which indicates the desirability of placing one box on top of another (Pheromone Trail). Each ant starts with an empty stack and iteratively selects the next box to place based on: Pheromone levels (favouring edges with more pheromone) and Heuristic information (ex: the height of the box and base area constraints). After all ants have constructed a solution, the pheromone levels are updated. The pheromone on edges used in the best solutions are increased, and some pheromones are evaporated too, to prevent stagnation. The process is repeated for a fixed number of iterations or until no significant improvement is observed.
This approach makes finding high-quality solutions easier, by balancing exploration and exploitation.

There are several real-life examples of the Box Stacking problem too. Any place or situation where there is organising or optimising boxes/containers/supplies or similar structures, there is some variation of the Box Stacking problem at play. This can be observed in Freight Storage, Warehouse Management, Architecture Business, Manufacturing and Packaging, Event and Space Planning, Disaster Management/Relief Operations, Commerce Centres, Aviation and Cargo Planning, Inventory Routing, Space Exploration and Satellite Design, Furniture Design and Home Organization, Sports Equipment, Shipping Operations, Art and

Museum Exhibits, Recycling/Waste Management, Energy and Resource Storage, Hospital and Medical Supply Management etc. The list goes on and on.

Conclusion:
The Box Stacking Problem, while seemingly simple, presents elaborate challenges that intersect with critical fields like logistics and optimization. By analysing and comparing different solutions and approaches, we have tried to emphasize the importance of algorithmic resourcefulness in addressing real-world concerns. This study aims to inspire further research and application of these methods and techniques in both theoretical and practical domains.

Thank you.